4 Static Operators

4.1 Basic Principle

- If a double arrow => is used instead of a single arrow -> in the declaration of an operator, the declared operator is a *static* operator.
  (Operators defined with a single arrow are also called *dynamic* operators in contrast.)

- A static operator has a “memory” to “remember” all invocations performed so far plus the corresponding result values.

- If it is invoked once more with the same syntax and the same parameter values as some time before, it directly returns the former result value from its memory and, therefore, the implementation of the operator is not evaluated again.

- This can be used in principle to automatically optimize operators in a table-based manner (dynamic programming). However, the number of different invocations should not be too large in that case.

- Much more important, however, is the possibility to use this to implement user-defined data structures.

4.2 Operators without Explicit Implementation

- If no explicit implementation is given in the declaration of a (static or dynamic) operator, the operator's implementation is implicitly an expression that returns a new (and thus unique) synthetic value in each evaluation (just like the operator `uniq` in § 3.7.3).

- But because the implementation of a static operator is not evaluated if it is invoked with the same syntax and the same parameter values as some time before, the same value as before is returned in such a case.

4.3 Open Types

4.3.1 First Idea

```
Point : type;

(p:Point) "@" x => (int?);
(p:Point) "@" y => (int?);

p1 : Point; p1@x =! 3; p1@y =! 4;
p2 : Point; p2@x =! 5; p2@y =! 6;

(p:Point) "." x -> (int = ?p@x);
(p:Point) "." y -> (int = ?p@y);

print only p1.x; print only ' '; print p1.y;     $$ 3 4
print only p2.x; print only ' '; print p2.y     $$ 5 6
```

Explanations

- `Point` is a constant of the meta type `type` with a new synthetic value and, therefore, represents a new unique type.
- For some value `p` of this type `Point`, `p@x` and `p@y` each returns a synthetic value of type `int?`, i. e., a variable with content type `int`.
- Because both are static operators, a new variable is returned if one of these operators is invoked the first time for a particular point `p`, but the same variable is returned for all further invocations of the respective operator for the same point `p`.
- `p1` and `p2` are constants of type `Point`, each having a new synthetic value, and, therefore, represent unique objects of type `Point`.
- Therefore, each of the expressions `p1@x`, `p1@y`, `p2@x`, and `p2@y` returns a different variable, which is the same for each evaluation of the respective expression, however. Therefore, this variable can be used to store the respective coordinate of the respective point.
- The expressions `p.x` and `p.y` are just abbreviations for `?p@x` and `?p@y`, respectively.
- Because the content of the variables `p@x` and `p@y` may change between invocations of `p.x` and `p.y`, these operators must be dynamic.

4.3.2 Generalization and Improvement

Generic Definitions

```
(U:type) "->" (V:type) => (type);

[(U:type) (V:type)]
(u:U) "@" (a:U->V) => (V? = u /\ a /\ v:V?);

[(U:type) (V:type)]
(u:U) "." (a:U->V) -> (V = ?u@a);

excl
  U : type; u1 : U; a1 : U -> U; u2 : U; a2 : U -> U; v : int?;
  (u1@a1) <-> (?v); u1 <-> v; a1 <-> v;
  (u2.a2) <-> (?v); u2 <-> v; a2 <-> v
end
```

Specific Usage

```
Point : type;
x : Point -> int;
y : Point -> int;

p1 : Point; p1@x =! 3; p1@y =! 4;
p2 : Point; p2@x =! 5; p2@y =! 6;

print only p1.x; print only ' '; print p1.y;     $$ 3 4
print only p2.x; print only ' '; print p2.y      $$ 5 6
```

Explanations

- For two arbitrary types `U` and `V`, `U->V` always returns a unique type that is used to represent *attributes* of type `U` with *target type* `V`.
- For an object `u` of any type `U` and an attribute `a` of this type with target type `V`, `u@a` normally (if `u` and `a` are not nil) returns a unique variable `v` with content type `V` in each case, which is used to store the value of attribute `a` of object `u`.
- Exception: If `u` or `a` is nil, `u@a` also returns nil (i. e., a nil variable) causing assignments to `u@a` to have no effect and `?u@a` to return nil in turn (cf. § 2.7; the conjunction `•/\•` whose right operand is evaluated only if necessary is defined on a task sheet).
- `u.a` is again an abbreviation for `?u.a`.
- The operators `•@•` and `•.•` shall have the same binding properties as the predefined variable query.
- In the specific usage, `x` and `y` are constants of the type `Point->int` with unique values and, therefore, represent two distinct attributes of type `Point` with target type `int`.
- Therefore, expressions such as `p1@x`, `p2@y`, `p1.y`, etc. have the same meaning as before.

4.3.3 Generic Constructor and Attribute Update Operator

Generic Definitions

```
$$ Create new object of the open type U
$$ and store v1, v2, ... as values of the attributes a1, a2, ...
(U:type) "(" [(V1:type)] (a1:U->V1) "=" (v1:V1)
        { "," [(V2:type)] (a2:U->V2) "=" (v2:V2) } ")" -> (U =
  u : U;
  u@a1 =! v1;
  { u@a2 =! v2 };
  u
);

$$ Store v1, v2, ... as new values of the attributes a1, a2, ...
$$ of object u.
[(U:type)] (u:U) "(" [(V1:type)] (a1:U->V1) "=" (v1:V1)
        { "," [(V2:type)] (a2:U->V2) "=" (v2:V2) } ")" -> (U =
  u@a1 =! v1;
  { u@a2 =! v2 };
  u
)
```

Specific Usage

```
$$ Create point p with coordinates 1 and 2.
p := Point(x = 1, y = 2);

$$ Set coordinates of p to 5 and 6.
p(y = 6, x = 5)
```

4.3.4 Generic Open Types

Example: Lists

```
$$ Generic Type T*
$$ for the representation of lists with element type T.
(T:type) "*" => (type);

$$ * shall bind stronger than ->
$$ because T -> T* shall be interpreted as T -> (T*).
excl (int -> int)* end;

$$ Generic attributes head and tail of T*.
[(T:type)] head => (T* -> T);
[(T:type)] tail => (T* -> T*);

$$ List with first element h and optional rest list t.
[(T:type)] (h:T) "->" [(t:T*)] -> (T* =
  T*(head = h, tail = t)
);
```

```
$$ -> shall bind stronger than constant declaration
$$ because ls := 1 -> should be interpreted as ls := (1 ->).
excl (x := 1) -> end;

$$ Length of the list ls.
[(T:type)] "#" (ls:T*) -> (int =
  p : T*?;
  p =! ls;
  while ?p do p =! ?p.tail end
)
```

Exemplary Use

```
$$ Create lists with different element types
$$ and print their lenghts.

ls := 1 -> 2 -> 3 -> 4 -> 5 ->;
print #ls;                          $$ 5

ls2 := 'a' -> 'b' -> 'c' ->;
print #ls2                          $$ 3
```

Explanations

- Because the type of the attributes `head` and `tail` depends on their type parameter `T`, these attributes cannot be defined as constants but rather as static operators.

- In typical applications of `head` and `tail`, the assignment of `T` and therefore the specific type of the attribute is always deduced from the application context (cf. § 3.7.3).

Output of Lists

```
$$ Print the elements of the list ls,
$$ provided there is an output operator for its element type T.
[(T:type)] print <o>[only] (ls:T*)
                  [(+ print only (T) -> (bool))] -> (bool =
  p : T*?;
  p =! ls;
  while ?p do
    print only ?p.head;
    print only '-'; print only '>';
    p =! ?p.tail
  end;
  <o>[true | print 1:0]
)
```

Exemplary Use

```
print ls2;                  $$ a->b->c->

ls3 := ls2 -> ls2 ->;
print ls3                   $$ a->b->c->->a->b->c->
```

4.3.5 Remarks

- Because additional attributes can always be added to a previously defined type later – also at different locations of a program or even in different modules –, these types are open for later extensions and are, therefore, called *open types*.

- Furthermore, not all objects of such a type must always have values for all attributes of the type. Then, non-existent attributes also do not occupy memory, and querying them always returns nil.

- Therefore, open types are very well-suited (in particular, much better than `union` types in C) for the flexible storage of variant data structures.

4.4 Array-like Types

Generic Definitions

```
$$ Generic type T []
$$ for the representation of arrays with element type T.
(type) "[" "]" => (type);

$$ Access to the i-th element of the array a.
[(T:type)] (a:T[]) "@" (i:int) => (T? = a /\ i /\ (v:T?));
[(T:type)] (a:T[]) "." (i:int) -> (T = ?a@i)
```

Specific Usage

```
letters : char [];
letters@1 =! 'a';
letters@26 =! 'z';

print letters.1;                $$ a
print letters.2;                $$ (nothing)
print letters.26                $$ z
```

4.5 Strings

- MOSTflexiPL deliberately does not provide a predefined type for strings, because a suitable type can easily be defined by any user himself (typically in a library).
  For example, the list type `char*` (cf. § 4.3.4) or an array-like type (cf. § 4.4) can be used for that purpose.
- Nevertheless, there are predefined *string literals* consisting of any number of Unicode characters inside of (double) quotation marks.
  Just as with names of constants (cf. § 2.4) and operators, such a quotation mark must be duplicated to include it into a string literal.
  (The only difference to such names is the fact, that string literals might also contain white space and might be empty.)
- To use such a literal, there must be an operator of type

  ```
  str { (c:char) } -> (S)
  ```

  with an arbitrary result type `S` in the current context, which is implicitly passed to the literal (cf. § 3.11) and will be invoked by it with all characters of the literal.
- The result value of the literal will then be the value with type `S` returned by this operator (i. e., string literals will then have type `S`).

Example

```
str { (c:char) } -> (char* =
  h : char*?;
  t : char*?;
  {
    t =! ?t@tail =! char*(head = c);
    ?h \/ h =! ?t
  };
  ?h
)
```

- The operator `str` constructs a list containing the characters `c` by appending each character at the end of the already constructed list and finally returns the start of the list.

Exemplary Use

```
print "Hello!";          $$ H->e->l->l->o->!->

print [only] (s:char*) -> (bool =
  s : char*?; s =! s;
  while ?s do
    print only ?s.head;
    s =! (?s.tail)
  end;
  [ true | print 1:0 ]
);

print "Hello!"           $$ Hello!
```

- Because the operator `str` that is implicitly passed to string literals has result type `char*`, string literals such as `"Hello!"` also have that type.
- Therefore, the operator `print` in the first line of the example denotes the output operator for lists defined at the end of § 4.3.4.
- The operator `print` defined afterwards prints only the characters of the list `s` without arrows. Because it is more specific than the generic output operator for lists, it is preferred in the last line of the example.